Code Search API, Base of Parallel Code Refactoring System For Safety Standards Compliance

Peter Jurnečka, Petr Hanáček and Matej Kačic

FIT BUT, Bozetechova 1/2 Brno, Czech Republic, {ijurnecka, hanacek, ikacic}@fit.vutbr.cz, www.fit.vutbr.cz

Received 3 February 2014; Accepted 27 April 2014;
Publication 2 June 2014

Abstract

New technologies of multi-core and massively parallel processors are becoming common parts of today's desktop computers. These state-of-the-art technologies allow programming of parallel applications and systems, however, creating parallel applications puts higher demands on programmers' skills, project maintenance and modification of existing source codes. Program flaws entered on source codes could have fatal consequences, specifically in aviation or medicine systems, due to possible fatal impacts in case of systems failure.

This paper describes the current status of aviation and medicine software safety standards, points out the common requirements of all these standards, specially the requirement for reliability. Reliability can be easily achieved using design patterns with verified reliable source code modules. In our research, we propose system for implementation of concurrency and synchronization design patterns into existing code. We have created parallel source code search API which is described in this paper, and which is planned to be used in our parallel code refactoring system for safety standards compliance. This API enables us to define appropriate places in source codes for introduction of parallel design patterns into existing parallel source codes. In next design iteration, the proposed system will provide suggestions of refactoring operations of found source codes, based on static code analysis and formal description of parallel design patterns.

Journal of Cyber Security, Vol. 3 No. 1 , 47–64.
doi: 10.13052/jcsm2245-1439.313

Keywords: software safety, parallel design patterns, code searching.

1 Introduction

Parallel or multithreaded applications are becoming more widespread. New technologies such as multicore processors and massively parallel processors of graphics cards have become widely available and usable in desktop computers. However, programming of parallel systems puts higher demands on the skills of programmers, and greater demands are also by the maintenance and modification of existing projects.

Area in which any mistake can have fatal consequences is aviation or medicine. Aviation safety standards [1, 2] play an important role, because failures may have fatal impact. When we speak about software in aviation, we mean software for avionics, which is a term used for electronic systems used in airborne environments, derived from words aviation and electronics. Examples of avionic systems used in aircrafts are flight control systems (autopilot), navigation systems or anticollision systems. Safety of the software is part of a whole system safety.

The purpose of the Food and Drug Administration (FDA) software validation standards [3, 4] is to consider its applicability to the validation of medical device software. The standards recommend an integration of software life cycle management and risk management activities. The software developer should determine the specific approach and level of effort to be applied based on these standards. On the other hand, FDA validation standards do not recommend any specific life cycle model or specific technique.

Avoiding mistakes is the goal of software standards in these areas. Other way to avoid mistakes is to facilitate the work of programmers by using design patterns and refactoring. Currently, much research has been done in the field of design patterns and refactoring of existing source code. However, the research of automated refactoring has not addressed design patterns of parallel and distributed systems.

Common requirement of all these standards is the requirement for reliability which can be achieved with design patterns. In our research, we propose system for implementation of parallel design patterns in existing code. The proposed system provides suggestions of refactoring operations based on static code analysis with code search API and formal description of parallel design patterns.

The main idea of proposed system is to use formally specified parallel design patterns in suggesting refactoring operations in editing of source

code. The aim of our research is to create a system that automatically assist the programmer in source code refactoring in implementing parallel design patterns into existing parallel source code. Source code created with use of design patterns is more efficient, easily manageable and therefore more reliable. To create such system we must combine parallel design patterns, refactoring and static code analysis.

2 Code Searching Problem

The issue of searching source codes has been given a great amount of research. There are different approaches used, each approach has its advantages, but in our context of the definition of insertion places of design patterns neither cannot be used, because none of these existing solutions do not search parallel source code and therefore has no information about access of program threads to each source code statement. This is our main contribution delivered by this article.

First existing code search solution is XL C++ Browser from [5], which is a distributed static analyzer for the C++ programming language. Key features of this technology are its support for semantic queries - queries that make use of the C++ semantics to interpret information about programs. It uses rules for describing the relations between the program symbols and it has capability to browse remote databases across network.

Steven P. Reis in his Semantics Based Code Search [6] describes his system which uses the vast repositories of available open source code to generate specific functions or classes that meet user specifications. He lets users specify what they are looking for as precisely as possible using keywords, class or method signatures, test cases, contracts, and security constraints. His Code Search system then uses an open set of program transformations to map retrieved code into what the user asked for.

This approach was implemented in prototype system for Java with web interface. Limitation of this solution is, that it is very tightly bound to java and generalization of search engine to other languages and implementation of thread info is more difficult than creating a new API.

Lemos in his article Applying TestDriven Code Search to the Reuse of Auxiliary Functionality [7] states that software developers spend considerable effort implementing auxiliary functionality used by the main features of system (e.g. compressing/decompressing files, encryption/description of data, scaling/rotating images). With the increasing amount of open source code available on the Internet, time and effort can be saved by reusing these utilities

through informal practices of code search and reuse. However, when this type of reuse is performed in an ad hoc manner, it can be tedious and errorprone: code results have to be manually inspected and extracted into the workspace. In his paper he introduces the use of test cases as an interface for automating code search and reuse and evaluate its applicability and performance in the reuse of auxiliary functionality. He calls his approach TestDriven Code Search (TDCS). Test cases serve two purposes: (1) they define the behavior of the desired functionality to be searched, and (2) they test the matching results for suitability in the local context. He presents CodeGenie, an Eclipse plugin that performs TDCS.

CodeGenie is most similar to the proposed solution, however as discussed above, CodeGenie does not search parallel source code and does not include information about threads and their access to source code statements.

Last found solution is Sourcerer [8]: Search Engine for Open Source Code Supporting Structure Based Search: The paper [8] focuses on the current research goals and search capabilities of Sourcerer. Sourcerer enables searches that are based not just on keywords but also on the structural properties and relations among program elements. Current version of Sourcerer works with the open source projects implemented in Java. The first release of Sourcerer, as of time of submission of that paper, is publicly available in its development version at http://sourcerer.ics.uci.edu/.

Sourcerer is closest to our approach. But because it uses large relational database as data storage and is closed only to Java and also has no information about threads, it is not usable in our context. Therefore we propose own code search API and system, which will be used to define the search queries for parallel source code. This queries will find suitable places in the code which we can propose to add a design pattern.

3 Our Solution

Our Code Search API provides interface for easily searching for specific code. It is one of base parts of our parallel code generating and refactoring system for safety standards compliance, which consists of two parts, design pattern code generator and the new code structure proposer. The code generator generates source code from existing source code using formally described design patterns. The code structure proposer founds appropriate places in code for applying design patterns using Code Search API and selected design patterns are then used to generate new source code containing code snippets from the original code.

As mentioned before, Code Search API provides interface, which is used for queuing existing source codes. This interface is used in our proposed design pattern specification language. Each design pattern is described with pair (*prec, spec*) where prec stands for precondition and spec for pattern specification.

Precondition defines places, where concrete design patterns should be placed. This is done using our proposed Code Search API, which provides robust language for static parallel code analysis used for determining appropriate design pattern usage. Whole static code analysis is running on our code searching framework which provides easy access to all classes, functions, properties in project including thread usage information for each of theese statements.

Pattern specification (spec) uses XML which consists of two main parts: entities (denoted by <entities>) and relations between them (denoted by <relations>). Part <entities> contains a definition of entities that exist in design patterns. Each entity can represent class, method, variable or property. Entities may contain attributes which determine the connection of the design pattern specification *spec* with each statement from the set of preconditions *prec*.

Main idea behind our Code Search API is extension of abstract syntax tree with thread usage information. This thread info contains information about threads for each line of code in source files. Source code is parsed into syntax tree and during this iteration, *ThreadStartFilter* filter is applied on each statement. If selected statement creates new thread, then new thread is added into Thread List. We have implemented C# prototype, and for example, our ThreadStartFilter returns all known thread starting statements (e.g. *Thread.Start(), Task.Factory. CreateAndStart(), Background Worker*).

4 Metamodel used for Modelling AST

There are multiple approaches for storing source code in a queryable repository, some are based on abstract syntax trees (ASTs) and others on relational databases. We did not want to create an entirely new metamodel for Code SearchAPI, but rather extend an existing one that met our requirements. It had to be sufficiently expressive as to allow structure-based analyses, and it had to be efficient and scalable enough to include thousands lines of source code. We settled on an adapted version of Ossher et al.'s [10] SourcerrerDB which was based on Chen et al.'s [11] C++ entity-relationship (ER) metamodel. While in principle as expressive as an DB-based metamodel, our object metamodel is better suited to define search queryes which are a cornerstone

of CodeSearchAPI. In addition, we agreed with their decision to focus on what they termed a top-level declaration granularity, as it provides a good compromise between the excessive model size of finer granularities and the analysis limitations of coarser ones. The metamodel we present here is an extended and modified version of Sourcerer metamodel [10]. As shown on Figure 1. our revised metamodel adds support for thread information and removes Java bindings, which are not usable in our environment. This metamodel is used to model the structure and reference information extracted from .Net C# projects. Each source code file contains the entities defined within it, the relations originating from those entities, and the comments associated with them.

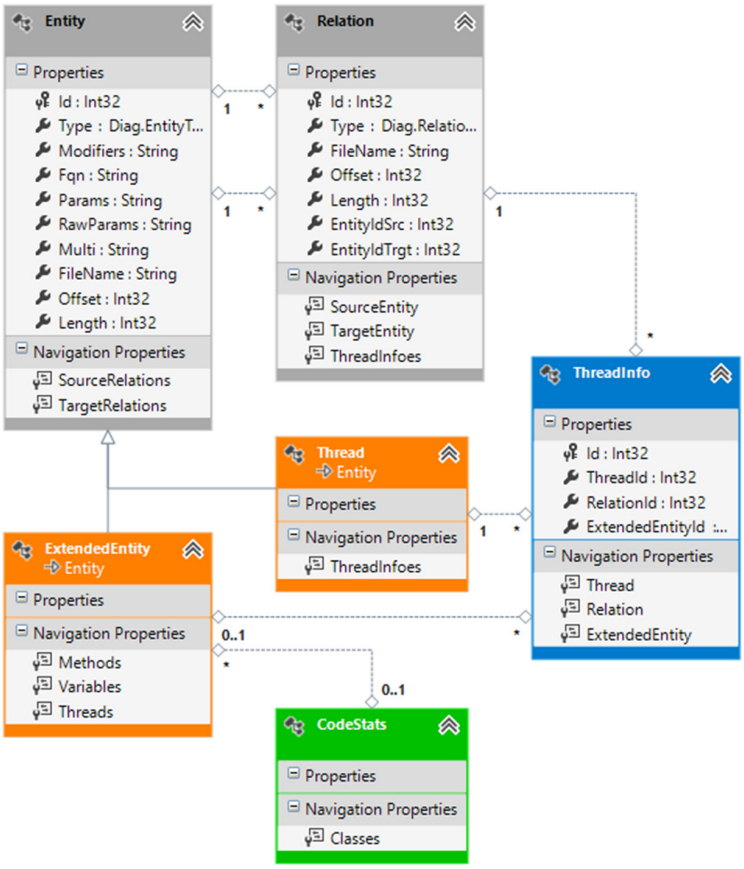


Figure 1 Extended syntax tree with thread and type info, collected by Code Search API

Base of our system is taken part of Sourcerer DB metamodel: table entites and relations. Our classes based on theese tables are created using Sourcerers proposed algorithms. On this basis, we have build our extended data structure, which contains classes: Extended Entity, Thread, Thread Info. Code Stats class serves as common unique entry point for all queries, and enables simple unified approach to search queries creation. The folowing paragraphs describe each class of our system, their creation and their purpose in our system.

4.1 Entity Class

Class Entity is taken from the design of Sourcerer. However, in our solution, we have removed specific java entity types ajd java bindings unusable in our system and we also have simplified the system for linking entities to files. The majority of entity types used in our metamodel correspond to explicit declarations in the .Net C# source code and shoul also be enough for all major object oriented languages. The entity types are: NAMESPACE, CLASS, INTERFACE, ANNOTATION, FIELD, INITIALIZER, CONSTRUCTOR , METHOD, PARAMETER, LOCAL VARIABLE, ARRAY, TYPE, PRIMITIVE, ENUM, ENUM CONSTANT, INSTRUCTION. Each entity is uniquely identified by its Fully Qualified Name (Fqn : String), file that it comes from, and its location in that file. Each entity is further annotated, when appropriate, with its modifiers (such as public or static).

4.2 Relation Class

Class Relation is also taken from the design of Sourcerer, however in our application we have removed relation types, which are not usable in our search queries. Table 1 contains the relation types in our metamodel. All of the relations are binary, linking a source entity with a target entity. A relation is identified uniquely by its type, and the FQNs of its source and target entities and source code location. As a result, any time the same relation is generated more than once, such as a method calling another method multiple times in loop in its body, those relations are collapsed into one.

4.3 Thread Class

Thread class is a special kind of Entity used to tracking of the threads in the application. Each thread is basically a special kind of method, which is called parallel with the main method of the program. As mentioned above, during the creation of AST the ThreadStartFilter is applied to each command and

Table 1 Relation Types

Relation	Description
CONTAINS	Physical containment
IMPLEMENTS	Interface implementation / extension
TYPE OF	Field type
RETURNS	Method return type
READS	Field access
WRITES	Field access
CALLS	Method invocation
INSTANTIATES	Constructor invocation
THROWS	Throws declaration or explicit throw
ANNOTATED BY	Annotation
USES	Any reference
PARAMETRIZED BY	Associated type variables
OVERRIDES	Function overrides

if it detects a new thread this thread is tracked using this special entity type Thread. For our purpose (search queries for automated insertion of paralle design patterns) we need to divide Entites into 3 groups: those never touched with any thread, those used with one thread and those used with two or more threads. This division simplifies the process of Thread detection. If some new thread is created in loop, or in recursive function call, we automaticly create two instances of new thread, because that is the worst case. As mentioned before, Thread as subclass of Entity is uniquely identified by its Fully Qualified Name and location in source code. When we during AST traversal come to already tracked location on source code, we create new Thread and continoue, only when there is only one Thread object created, Otherwise we skip this code as already covered. In this state of our research, we do not take into account thread differences, caused by parameters used by thread creation code.

4.4 ThreadInfo Class

ThreadInfo class is used for the definition of relation between Threads and Entities. They tell us that selected thread touches variable in this function call. ThreadInfo class are markes all Relatons associated with each line of thread source code line. ThreadInfo classes are created during the construction of Thread entities, by taking all Relations related to main thread function. If the main thread function calls other functions, also those relations are marked.

4.5 ExtendedEntity and CodeStats Classes

ExtendedEntity and CodeStats classes simplify the definition of search queries. CodeStats class is a single common entry point for all queries. ExtendedEntity classes create tree structure and are formed during the creation of Entities during first pass of code parsing and creation of AST. If the type of Entity is class or method or property the ExtendedEntity is created. ExtendedEntity class extends Entity with three additional navigation properties which are Methods, Variables and Threads. Property Methods is used only when ExtendedEntity points to class and contains references to all methods and constructors within class. Property Variables contains information about all shared variables within selected ExtendedEntity. Property Threads links ExtendedEntity with ThreadInfo and is its main purpose is to make querying easier. As we shoved in next chapter, this metamodel is sufficient for our proposed specification of design patterns and their automatic insertion into existing source codes.

5 Results

We have applied this system to set of six synchronization patterns from the POSA catalogue [9]. In almost all cases it is possible to find reasonable precondition. Next paragraphs provide short description, UML diagram and found preconditions of selected patterns described by Microsoft LINQ queries into our CodeSearch API data model. Possibility of definition of query using our CodeSearch API showes, that CodeSearch API can be used in next iteration of our code refactoring research. To give ourselves coarse outline, in next subchapters we also provide easily readable version of preconditions of selected synchronization design patterns. All these patterns are used for avoiding synchronization problems between threads.

5.1 Thread Safe Interface

As shown on Figure 2 Thread - Safe Interface pattern divides the functions of the component to the public available interface and private implementation methods. Public interface acquires the lock, calls corresponding private method and then releases the lock.

To ensure proper synchronization of client threads call functions of only the public interface. Interface function obtains necessary locks and calls the appropriate implementation of the function that no longer cares about locking and can freely call other implementation functions. Deadlock (selfdeadlock) cannot occur because the lock is obtained only once at the beginning in the

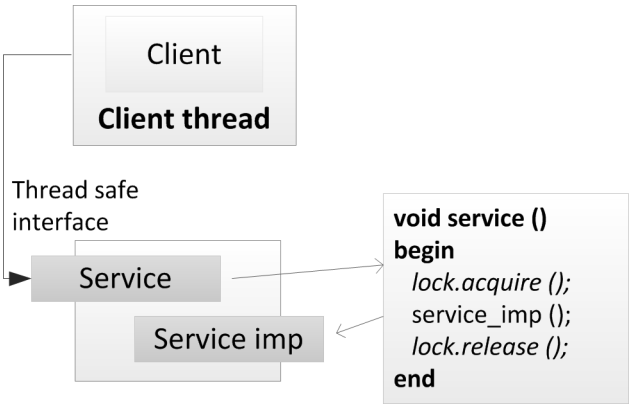


Figure 2 Thread - Safe Interface design pattern

```
CodeStats.Classes
.SelectMany(type => type.Functions)
.Where(func => ((func.Threads.Count > 1)
&& (GetLocks(func).Count() == 0));
```

Figure 3 Thread - Safe Interface precondition code

public interface and the recursive function calls do not obtain the lock more times, which also improves the performance of the application.

From this description we can define the preconditions of this pattern, which are: There exists an object with one or more functions or properties, which are used by more than one thread and this function or property does not acquire any lock. As we can see on next code snippet, this precondition uses ThreadInfo, from CodeSearch API, and function GetLocks(ClassMember x) which gets locks used by selected member of class. On Figure 3. we can see Microsoft LINQ source code version of precondition of this pattern.

5.2 Future

As shown on Figure 4 the Future pattern immediately after calling the constructor returns the "virtual" data object, called the Future. Future object contains information about the state and the calculation of the results is dispatched on service thread. The future object returns the result only if it is valid.

If the client thread wants to read the future value of the object before there is a valid result, the future object suspends client thread until a valid result

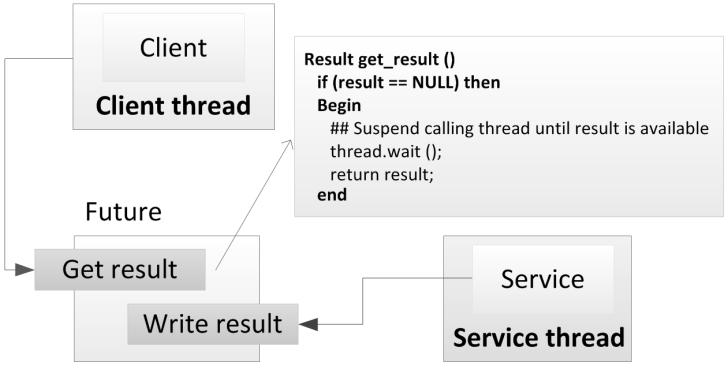


Figure 4 Future design pattern

```
CodeStats.Classes
.SelectMany(type => type.Functions)
.Where(func =>
 AvgFunctionLen(func) >= treshold);
```

Figure 5 Future design pattern precondition code

is written in the future object. Future object can also contain nonblocking function available for checking the validity of stored value.

From this description we can define the preconditions of this pattern, which are: There exists object X, this object contains function or parameter with execution time longer than user defined threshold. Or, there exists object X with constructor with execution time longer than user defined threshold, and first call of parameter or function of this object is far away from its construction. As we can see on next code snippet, this precondition uses function AvgFunctionLen(ClassMember x) which gets average function length of selected member of class, which is computed by counting executed instructions. For safety and calculability reasons AvgFunctionLen counts only to UInt16.MaxValue. On Figure 5. we can see Microsoft LINQ source code version of precondition of this pattern.

5.3 Guarded Suspension

As shown on Figure 6 Guarded Suspension design pattern instead of termination of the blocked functions suspends the thread, so that other threads can access shared component and thus change the value of the guard conditions and the release threads of the blocked functions.

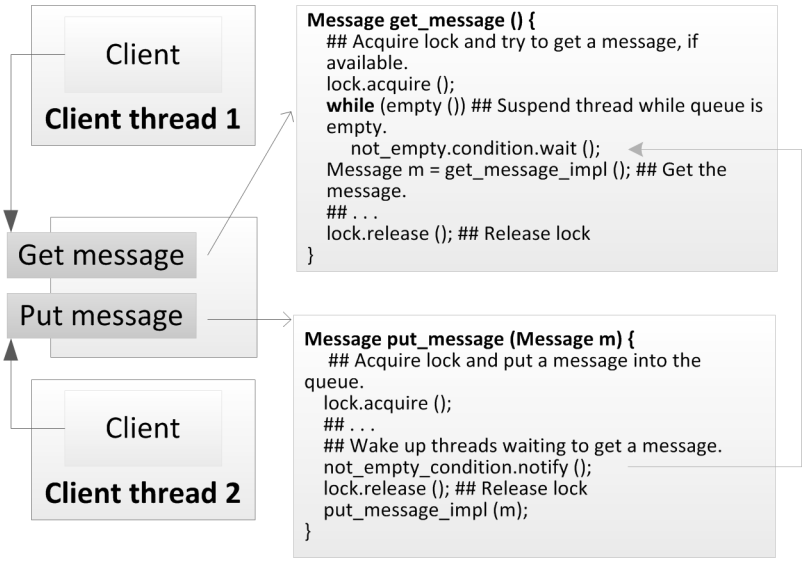


Figure 6 Guarded Suspension design pattern

The main contribution of Guarded Suspension design pattern is that it minimizes the costs associated with parallelization and also increases the availability of shared components. If threading model is designed to make suspension on OS layer, the price of the suspension and the synchronization is minimal.

From this description we can define the preconditions of this pattern, which are: There exists function that is dispatched in separate thread, and this function contains condition which aborts functions thread without any other computation. As we can see on next code snippet, this precondition uses function InstrCounter(ClassMember x) which creates pairs instructionId, instruction. For safety and calculability reasons InstrCounter counts only to UInt16.MaxValue. On Figure 7 we can see Microsoft LINQ source code version of precondition of this pattern.

```
CodeStats.Classes
.SelectMany(type => type.Functions)
.Select(func => InstrCounter(func))
.Where(instr =>
 (instr.Id <= treshold) && (instr.IsReturn));
```

Figure 7 Guarded Suspension design pattern precondition code

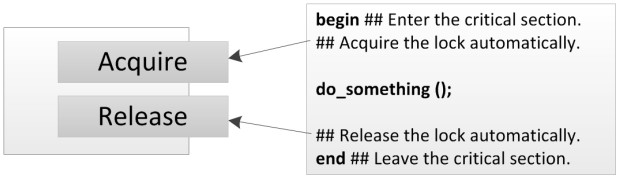


Figure 8 Scoped Locking design pattern

```
CodeStats.Classes
.SelectMany(type => type.Variables)
.Where(v => v.Threads.Count() > 1)
.Where(v => GetLocks(v).Count() == 0);
```

Figure 9 Scoped Locking design pattern precondition code

5.4 Scoped Locking

As shown on Figure 8 the Scoped Locking design pattern outlines the critical section with lock statement which automatically gets a lock at the entrance, and automatically releases the lock on any way from the lock frame. Scoped Locking design pattern increases the robustness of parallel software by eliminating common programming errors associated with synchronization of multiple threads. Locks are obtained automatically when the thread enters the critical section, and automatically released when it leaves out. Implementation of this design pattern depends on the programming language. For example, the Java programming language contains the synchronized keyword which instructs the compiler to automatically generate the appropriate instructions serving locking and unlocking locks.

From this description we can define the preconditions of this pattern, which are: Number of shared variables among the program is small, and access to these variables is not secured with any locks. As we can see on next code snippet, this precondition uses ThreadInfo, from CodeSearch API, and function GetLocks(ClassMember x) which gets locks used by selected member of class. On Figure 9 we can see Microsoft LINQ source code version of precondition of this pattern.

5.5 Immutable Value

As shown on Figure 10. Immutable Value design pattern defines the design objects whose instances are immutable. The internal state of an object is set in the constructor, and no further changes are allowed.

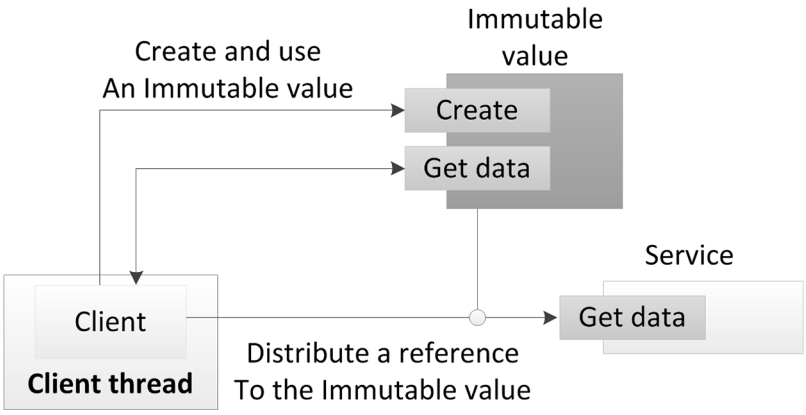


Figure 10 Immutable Value design pattern

```
CodeStats.Classes
.SelectMany(type => type.Variables)
.Where(v => GetWritesCount(v) == 1);
```

Figure 11 Immutable Value design pattern precondition code

In an immutable object are only read only parameters. The absence of any possibility of changing the object removes any need for synchronization and thus simplifies and improves efficiency the work of the program. By eliminating the need to copy objects we also improve program performance.

From this description we can define the preconditions of this pattern, which are: All variables, that are only read in program execution, or are written only one time. As we can see on next code snippet, this precondition uses function GetWritesCount(Variable x) which gets number of writes into this variable during program execution. On Figure 11 we can see Microsoft LINQ source code version of precondition of this pattern.

6 Conclusions

The importance of safety standards of software systems is increasing as the use of software grows because of its convenience and flexibility. Software safety standards are very important in aircraft, military, automotive or medical devices. Common requirement of all standards is reliability. Reliability can be easily achieved with design patterns. We are creating system for implementation of parallel design patterns into existing code. Our system will

provide suggestions of refactoring operations based on static code analysis and formal description of parallel design patterns. For this purpose, we have created custom CodeSearch API, which queries will be used in design patterns definitions, for defining appropriate places for design patterns suggestions. The last part of article gives us of an overview of application of created CodeSearch API on a set of six synchronization patterns from the POSA catalogue [9]. In all cases it was possible to find reasonable precondition using our Search API. In next iteration we focus our research on better customizable XML description language, an finalization of whole refactoring system, so we can then provide full robust parallel design patterns refactoring system for safety standards compliance.

This work has been supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and by BUT FIT grant FIT-S-11-1: "Advanced secured, reliable and adaptive IT" and by Research Plan No. MSM0021630528.

References

[1] Howard C. (2011). DO-178B safety certification and other software security tools drive avionics software designs. 2011. Available at: http://goo.gl/ZkzyF

[2] Federal Aviation Administration. Advisory Circular 20-115B. 1993. Available at: http://goo.gl/C6d1k

[3] General Principles of Software Validation; Final Guidance for Industry and FDA Staff, Available at: http://goo.gl/HjIKb

[4] Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices, Available at: http://goo.gl/JqkYr

[5] SHARMAN, J. ET AL. (1992) Architecture of the XL C++ browser, CASCON '92 Proceedings, P: 369–379, IBM Press

[6] REISS, P. STEVEN, (2009) Semantics-Based Code Search, ICSE 09 Proceedings, IEEE

[7] LEMOS, OTAVIO AUGUSTO LAYYARINI, ET AL. (2009) Applying Test-Driven Code Search to the Reuse of Auxiliary Functionality, SAC 09 Proceedings, ACM

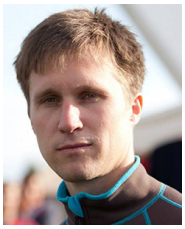
[8] SUSHIL B., ET AL. (2006) Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search, OOPSLA 06 Proceedings, ACM

[9] BUSCHMANN, F. ET AL. (2007) Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing. John Wiley & Sons, Inc., New York, NY USA, ISBN: 978-0-470-05902-9.
[10] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes, "SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects," in Proceedings of the International Workshop on Mining Software Repositories. Vancouver, Canada: IEEE Computer Society, 2009, pp. 183–186.
[11] Y.-F. Chen, E. R. Gansner, and E. Koutsofios, "A c++ data model supporting reachability analysis and dead code detection," IEEE Trans. Softw. Eng., vol. 24, no. 9, pp. 682–694, 1998

Biograpies

Peter Jurnečka. He received his M.Sc. from Brno University of Technology in 2009. He is currently a Ph.D. student at Faculty of Information Technology, Brno University of Technology. His research interests are in information technology security and safety, especially in using parallel design patterns for safety standards compliance.

Petr Hanáček. He graduated at Brno University of Technology. He is currently an Associate Professor at Faculty of Information Technology, Brno University of Technology. His research interests are in security of information systems, applied cryptography and wireless systems.



Matej Kačic. He received his M.Sc. from Brno University of Technology in 2010. He is currently a Ph.D. student at Faculty of Information Technology, Brno University of Technology. His research interests are in information technology security, especially in wireless systems.